Methods:

Defining Classes, Methods & Variables

In this lecture, we will go through some methods from Standard Ruby Library.
You will learn how to define classes, modules, methods and all kinds of variables.

Creating Classes / Modules

```ruby
# self == main

class ClassName
    # Class definition body

    # self == ClassName
end


class ClassName < InheritFrom

end


module ModuleName
    # Your module definition body
end
```

```ruby
# self == main

ClassName = Class.new do
    # Class definition body

    # self == #<Class:0x1ed9488>
end


ClassName = Class.new(InheritFrom) do

end


ModuleName = Module.new do
    # Your module definition body
end
```

Scope isn't changing, so what?

```ruby
local_var = [:foo, :bar, :baz]

MyClass = Class.new do
    local_var.each { |var| attr_accessor var }

    # This will not produce our 3 methods:
    #   local_var.each do |var|
    #       def var
    #           instance_variable_get :"@#{var}"
    #       end
    #   end
end

MyClass.instance_methods(false)                       # [:foo, :foo=, :bar, :bar=, :baz, :baz=]
```

Task / exercise:

We are given an array of symbols (local_var). For each value in the array, we want to create a method with that value and the method must return the value of instance variable with the same name.

Basically, we want this result:

```
    local_var.each { |name| attr_reader :name }
```

How can we define methods?

- Using usual "def" syntax
- define_method
  (Defines instance method in the receiver. It's a private method.)
  define_singleton_method
  (Defines singleton method in the receiver.)

Using usual "def" syntax

```
class MyClass
    def instance_method
        @instance_variable = ""
    end

    def self.class_method(a, b, c)
        @class_instance_variable = a
    end
end
```

```
MyClass = Class.new do
    def instance_method
        @instance_variable = ""
    end

    def self.class_method(a, b, c)
        @class_instance_variable = a
    end
end
```

```
MyClass.instance_methods(false)                       # [:instance_method]
MyClass.methods(false)                                # [:class_method]
MyClass.singleton_class.instance_methods(false)       # [:class_method]
```

Task: Define methods from array

```ruby
local_var = [:foo, :bar, :baz]


MyClass = Class.new do

    local_var.each do |var|
        def var
            instance_variable_get :"@#{var}"
        end
    end

end

MyClass.instance_methods(false)                                      # [:var]
```

define_method syntax

```ruby
define_method(:name, [Proc, Method, UnboundMethod])


define_method(:name) do
    # Method definition body
end
```

define_method(:name) { }

```ruby
MyClass.define_method(:foo) { }                            # NoMethodError: private method `define_method' call


class MyClass
    define_method :foo do |arg|
        "MyClass#foo( #{arg} )"
    end
end

MyClass.new.foo(42)                                        # MyClass#foo( 42 )


class << MyClass
    define_method :bar do
        "MyClass.bar"
    end
end

MyClass.bar                                                # MyClass.bar
```

Task: Define methods from array

```ruby
local_var = [:foo, :bar, :baz]

MyClass = Class.new do

    local_var.each do |var|
        define_method var do
            instance_variable_get :"@#{var}"
        end
    end

end

MyClass.instance_methods(false)                                    # [:foo, :bar, :baz]
```

define_singleteon_method syntax

```
define_singleton_method(:name, [Proc, Method, UnboundMethod])


define_singleton_method(:name) do
    # Method definition body
end
```

define_singleton_method(:name) { }

```ruby
obj = MyClass.new

obj.define_singleton_method :foo do
    "obj#foo"
end

obj.foo                                                       # obj#foo



MyClass.define_singleton_method :bar do
    "MyClass.bar"
end

MyClass.bar                                                   # "MyClass.bar"
```

How can we define methods?

- Using usual "def" syntax
- define_method
  (Defines instance method in the receiver. It's a private method.)
  define_singleton_method
  (Defines singleton method in the receiver.)

Task: Define methods from array

```ruby
local_var = [:foo, :bar, :baz]


MyClass = Class.new

# Since there is no way to use `define_singleton_method` to
# create instance methods on a class. There is no solution
# to our task using `define_singleton_method` method.
```

How can we define variables?

- Using usual syntax
  (Doesn’t work with all variable types as you would expect.)
- Using methods from Standard library
  (instance_variable_set, class_variable_set, const_set, local_variable_set)

Defining variables: Using usual syntax

```ruby
class MyClass

    @@class_var = ""

    CONST = ""

    @eigen_instance_var = ""

    def initializer
        @instance_variable = ""
    end

end
```

```ruby
MyClass = Class.new do
    # warning: class variable access from toplevel
    @@class_var = "top level"

    CONST = "top level"

    @eigen_instance_var = ""

    def initialize
        @instance_variable = ""
    end

end
```

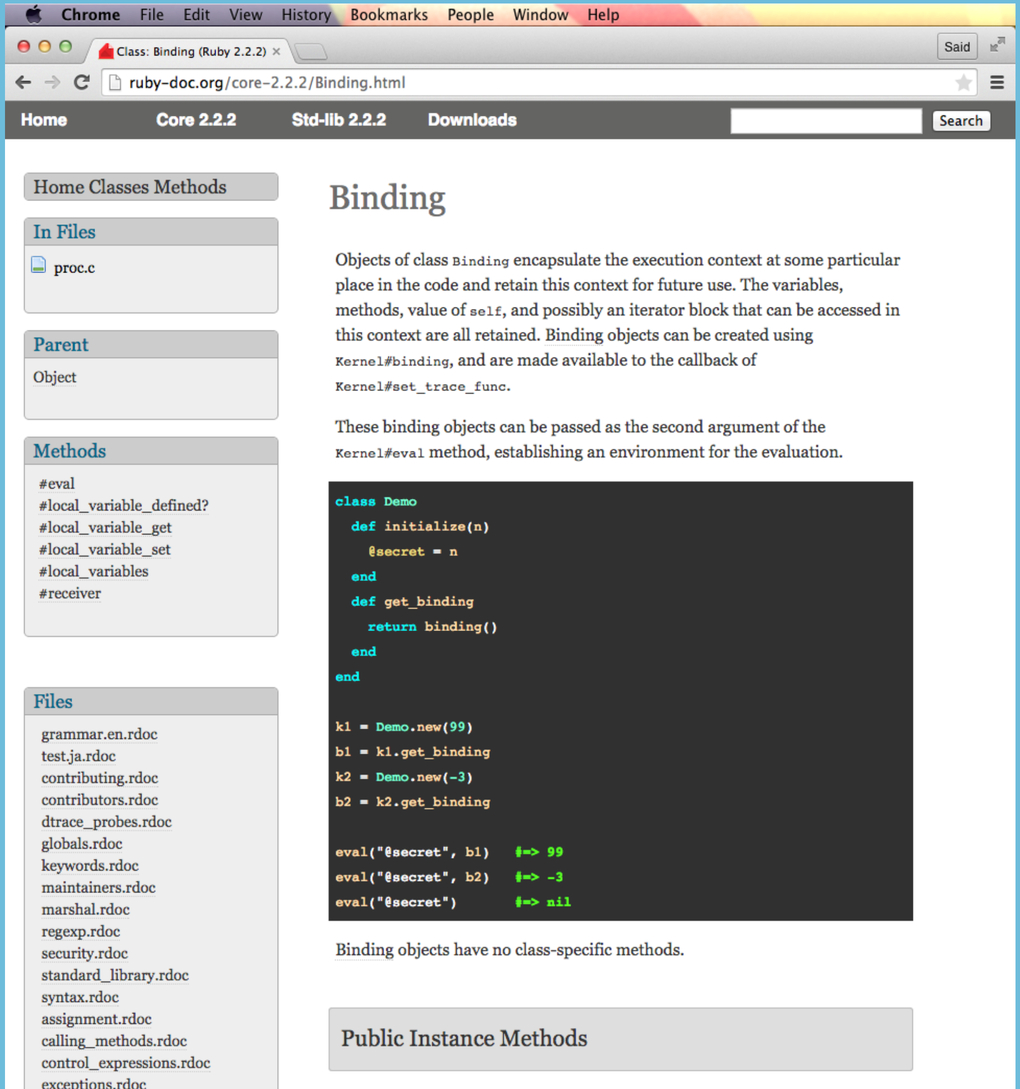
Defining variables: Using usual syntax

```ruby
MyClass = Class.new do
    # warning: class variable access from toplevel
    @@class_var = "top level"

    CONST = "top level"

    @eigen_instance_var = ""

    def initialize
        @instance_variable = ""
    end

end


self                                                        # main
self.class                                                  # Object

Object.class_variables                                      # [:@@class_var]
Object.constants.include?(:CONST)                           # true

MyClass.instance_variables                                  # [:@eigen_instance_var]
MyClass.new.instance_variables                              # [:@instance_variable]
```

Defining variables: Using std lib methods

```
MyClass = Class.new do
    # warning: class variable access from toplevel
    @@class_var = "top level"

    CONST = "top level"

    @eigen_instance_var = ""

    def initialize
        @instance_variable = ""
    end

end


MyClass.const_set :CONST2, 'foo'
MyClass.constants                                             # [:CONST2]

MyClass.class_variable_set :@@class_var, 'foo'
MyClass.class_variables                                       # [:@@class_var]


obj = MyClass.new
obj.instance_variables                                        # []
obj.instance_variable_set :@inst_var, 'foo'
obj.instance_variables                                        # [:@inst_var]


MyClass.instance_variables                                    # [:@eigen_instance_var]
MyClass.instance_variable_set :@eigen_inst_var, 'foo'
MyClass.instance_variables                                    # [:@eigen_instance_var, :@eigen_inst_var]
```

#1 Source — Ruby Documentation


Chrome File Edit View History Bookmarks People Window Help
Class: Binding (Ruby 2.2.2)
Said
ruby-doc.org/core-2.2.2/Binding.html
Home Core 2.2.2 Std-lib 2.2.2 Downloads
Search
Home Classes Methods

In Files

proc.c

Parent

Object

Methods

#eval
#local_variable_defined?
#local_variable_get
#local_variable_set
#local_variables
#receiver

Files

grammar.en.rdoc
test.ja.rdoc
contributing.rdoc
contributors.rdoc
dtrace_probes.rdoc
globals.rdoc
keywords.rdoc
maintainers.rdoc
marshal.rdoc
regexp.rdoc
security.rdoc
standard_library.rdoc
syntax.rdoc
assignment.rdoc
calling_methods.rdoc
control_expressions.rdoc
exceptions.rdoc

Binding

Objects of class `Binding` encapsulate the execution context at some particular place in the code and retain this context for future use. The variables, methods, value of `self`, and possibly an iterator block that can be accessed in this context are all retained. Binding objects can be created using `Kernel#binding`, and are made available to the callback of `Kernel#set_trace_func`.

These binding objects can be passed as the second argument of the `Kernel#eval` method, establishing an environment for the evaluation.

```
class Demo
  def initialize(n)
    @secret = n
  end
  def get_binding
    return binding()
  end
end

k1 = Demo.new(99)
b1 = k1.get_binding
k2 = Demo.new(-3)
b2 = k2.get_binding

eval("@secret", b1)   #=> 99
eval("@secret", b2)   #=> -3
eval("@secret")       #=> nil
```

Binding objects have no class-specific methods.

Public Instance Methods

List of classes where the methods are defined:

- http://ruby-doc.org/core/BasicObject.html
- http://ruby-doc.org/core/Object.html
- http://ruby-doc.org/core/Kernel.html
- http://ruby-doc.org/core/Module.html
- http://ruby-doc.org/core/Class.html
- http://ruby-doc.org/core/Binding.html